Generating Multiple Outputs from Ω

John Plaice
School of Computer Science and Engineering
The University of New South Wales
UNSW SYDNEY NSW 2052, Australia
plaice@cse.unsw.edu.au
http://www.cse.unsw.edu.au/~plaice

Yannis Haralambous
Département Informatique
École Nationale Supérieure des Télécommunications de Bretagne
CS 83818, 29238 Brest Cédex, France
Yannis.Haralambous@enst-bretagne.fr
http://omega.enstb.org/yannis

Abstract

In this paper, we describe how to generate multiple outputs (DVI, PostScript, PDF, XML, ...) from the same Ω document. The Ω engine is augmented with a library for manipulating multidimensional contexts. Each macro can be defined in multiple versions, and macros can thereby adapt to differing contexts. Macros can be specialized for several different output formats, without changing the overall structure. As a result, the same document can be used to easily produce different output formats, with appropriate specializations for each of them, without having to make any changes to the document itself.

Résumé

Dans cet article nous décrivons le processus de génération de sorties multiples (DVI, PostScript, PDF, XML, ...) à partir du même document Ω. Le moteur Ω a été muni d'une bibliothèque de sous-routines dédiée à la manipulation de contextes multi-dimensionnels. Les macros TEX peuvent être spécialisés selon le format de sortie, sans changer leur structure globale. Ainsi, le même document peut, sans la moindre modification, produire facilement différents formats de sortie avec les spécialisations ad hoc.

Introduction

We present in this paper a new approach to generating typeset and structural material from Ω in a number of different output formats. This approach generalizes the existing approaches of DVI postprocessors capable of interpreting DVI `\special`'s, specialized modifications to the typesetting engine, judicious use of alternate versions of macros, and external interpreters of subsets of LATEX.

Key to this new approach is the introduction in Ω of *versioned macros* and *versioned ΩTPs* that can adapt their behavior to a dynamically running tree-structured context that permeates the entire typesetting process. As a result, when a text is to be typeset for a new output format, then new versions of macros can be written at any level, *without changing the existing macros*, thereby minimizing the amount of additional work to be undertaken.

Versioned macros and ΩTPs have ramifications well beyond the structural issues involved in generating material for different output formats: versioning the typesetting process also provides a high-level interface for multilingual typesetting, an issue that has hindered the development of the Ω system since its inception. See the paper presented at *TUG* 2003, with Chris Rowley [3], for a detailed discussion.

However, it is not sufficient simply to be able to generate different versions of macros and ΩTPs; the TEX document model is very simple, and the one-pass document manipulation approach — analogous to the Pascal language in which it was written — built into the software acts like a straitjacket when one wishes to pass as input or to generate as output significantly different document structures.

Therefore, at least three additional components need to be added to Ω in order for it to be fully adaptable to different formats. First is the ability to directly apply ΩTPs and other filters to the input stream, even before, and possibly bypassing, the macro processing stage.

Second is the ability to directly apply ΩTPs to the output stream, possibly without even generating DVI output. Third is to supply general hooks that allow the user to manipulate internal document *structure*, and not simply horizontal and vertical boxes.

In this article, we present the work that we have undertaken towards these goals. We begin with a brief background, describing what we consider to be the main contributions of existing extensions of the general TEX framework (not just to the TEX engine), and show how these different approaches all contribute to a better understanding of the general problem of generating different outputs from the same files.

The model for contexts that we have adopted was developed by Paul Swoboda in his PhD thesis [7]. It is the most highly developed presentation of *intensional versioning*, an approach to the development of software variants first proposed by the first author and William W. Wadge [6]. We give a discussion of intensional programming and versioning, then give a detailed presentation of contexts and context operators.

We then show how contexts have been integrated into Ω. To do this, new Ω primitives are introduced for creating and using different versions of macros, and for changing and manipulating the runtime context. In addition, means for having versions of internal and external ΩTPs are defined.

These technical sections are followed by a discussion of how the internals of the Ω engine should be reorganized to facilitate the generation of multiple outputs.

TEX and its Extensions

The TEX document model supposes that a stream of text, interspersed with control sequences, is to be transformed into a series of *pages*, each of which is a vertical box that contains other boxes, either vertical or horizontal. Each page is generated into DVI output, in the process losing some of the information contained in the page.

The boxes are generated on the fly. Although certain items can be stored for later use in registers, accessed much as one would in assembly programming, TEX's document model essentially consists of the following processes:

- transforming streams of characters into streams of typeset glyphs and boxes (*main loop*);
- building math lists from TEX math, then transforming the lists into streams of typeset glyphs (*math mode*);
- transforming streams of typeset glyphs, with inserted hyphenation points, into streams of horizontal boxes, corresponding to lines (*paragrapher*);
- building boxes, corresponding to tables, from alignment specifications;
- building pages from streams of boxes and glue (*page builder*).

TEX's operation is undertaken in one pass, and it is very difficult, if not impossible, to be able to manipulate intermediate data structures as they are being built.

The different extensions to TEX and the different DVI postprocessors have all taken different approaches, which is quite normal given their divergent aims.

First are the DVI postprocessors, `dvips` (generating PostScript) and `dvipdfm` (generating PDF). Each of these programs transforms DVI output, augmented with DVI `\special`'s, specifically designed for use with that program and generated by TEX through its macro mechanism, into the relevant output format.

The main advantage of this approach is that it encourages modularity, in the sense that the typesetter is separate from the pretty-printer. However, one can only put into `\special`'s information that is made available to the user. Information about intermediate data structures is not directly available, so can only be approximated.

Second is the LATEX2HTML approach. This tool does not do typesetting, rather it reorganizes the structure of the text into HTML. It does not use the TEX engine, but itself parses a large (reasonable) subset of LATEX. For parts that cannot be directly translated into HTML, such as mathematics, then it generates small LATEX files, calls LATEX, then `dvips`, then transforms then into PNG files. Although LATEX2HTML is a useful tool, in its current form it will never have access to TEX's internal data structures, since it never calls TEX.

Third, also for generating HTML, is TEX4HT, which produces HTML files that resemble DVI pages generated by TEX. TEX4HT is also standalone, but it does use TEX for parsing and typesetting the input. It makes use of extensive DVI `\special`'s.

Fourth are the extensions to the TEX engine, namely e-TEX, pdfTEX, and Ω. The e-TEX extensions focus mainly on improving the macro expansion facilities. They do not change the typesetting, but do provide the very useful ability to reparse an input sequence.

The pdfTEX extensions are two-fold. First is some experimental work simulating some of Peter Karow's *hz* program. Second, more commonly used, are the extensions to generate PDF directly rather than DVI. In addition to its new pretty-printer for TEX pages, pdfTEX provides built-in mechanisms, using *whatsit* nodes, for generating such things as PDF forms and margin items.

Although pdfTEX is practical, in the sense that one can quickly generate PDF files from a TEX file, the fact that all of the functionality is hard-coded limits the possibility for extending the same system. For example, the current pdfTEX does not allow EPS files to be included in the PDF files that it generates.

The Ω extensions are of a more general nature.

In the Ω model, before glyph selection, the character stream to be typeset is segmented and processed by a series of filters, each reading from standard input and writing to standard output. Once all of the filters are applied, the stream is passed to the standard TEX character-level typesetter.

In addition, Ω also includes an experimental pretty-printer for MathML and XML. One of the goals of this work is to provide the means for recovering structure, particularly in mathematics expressions, in TEX and LATEX files that do not have perfect markup. However, this functionality is not in any way integrated with the ΩTP mechanism.

From this discussion, one can start to elucidate what is needed. One should be able to have, in a single system:

- internal data structures corresponding to page, paragraph, line, etc., that can be *explicitly* manipulated by the user;
- input mechanisms not based on the macro language that generate these data structures;
- more advanced macro-processing and other programming languages to manipulate these data structures;
- mechanisms to pretty-print the data structures in multiple output formats;
- limiting the output-format-specific additions to the internals of the typesetting engine.

For this whole approach to work, it is important that as one moves from one mechanism to another, changing input or output formats, and what is expected of the typesetting engine, that the whole framework be sufficiently *flexible* that one does not have to completely reprogram everything. In other words, the system must adapt to a complex context with many parameters. The next few sections focus in detail on how to deal with context in Ω; they are followed by a discussion of the points raised in the above wishlist.

Intensional Programming

Intensional programming [4] is a form of computing that supposes that there is a multidimensional context, and that all programs are capable of adapting themselves to this context. The context is pervasive, and can simultaneously affect the behavior of a program at the lowest, highest and middle layers.

When an intensional program is running, there is a *current context*. This context is initialized upon launching the program from the values of environment variables, from explicit parameters, and possibly from active context servers. The current context can be modified during execution, either explicitly through the program's actions, or implicitly, through changes at an active context server.

A context is a specific point in a multidimensional space, i.e., given a dimension, the context will return a value for that dimension. The simplest contexts are dictionaries (lists of attribute-value pairs). A natural generalization is what will be used in this paper: the values themselves can be contexts, resulting in a tree-structured context. The set of contexts is furnished with a partial order $\sqsubseteq$ called a *refinement relation*.

For example, to describe Australian English, we could use the context:

```
<script:<Latin>+
 lang:<English;dialect:<Australian>>>
```

where `script` and `lang` are called *dimensions*, and `lang:dialect` is called a *compound dimension*. See below for more details.

During execution, the current context can be queried, dimension by dimension, and the program can adapt its behavior accordingly. In addition, if the programming language supports it, then contextual conditional expressions and blocks can be defined, in which the *most relevant* case, with respect to the current context and according to the partial order, is chosen among the different possibilities.

In addition, any entity can be defined in multiple *versions*, which are mappings from contexts to objects. Whenever an identifier designating an entity appears in an expression or a statement, then the most relevant version of that entity, with respect to the current context, is chosen. This is called the *variant substructure principle*. The general approach is called *intensional versioning* [6].

The ISE programming language [5] was the first language combining both intensional programming and versioning. It is based on the procedural scripting language Perl, and it has greatly facilitated the creation of multidimensional Web pages. Similar experimental work has been undertaken under the supervision of the first author with C, C++, Java, and Eiffel. And, when combined with a context server (see Paul Swoboda's PhD thesis [7]), it becomes possible for several documents or programs to be immersed in the same context.

Structuring the Context

We use the same notation to designate contexts and versions of entities. This section has three subsections. First, we define contexts and the refinement relation. Then, we define *version domains*, which hold versioned entities. Finally, we define *context operators*, which are used to change from context to context. In the following section, we will show how all of these are to be used.

Contexts and Refinement. Let $\{(\mathbb{S}_i, \sqsubseteq_i)\}_i$ be a collection of sets of ground values, each with its own partial order. Let $\mathbb{S} = \cup_i \mathbb{S}_i$. Then the set of contexts $\mathbb{C}$ ($\ni C$) over $\mathbb{S}$

is given by the following syntax:

$$C ::= \bot \mid A \mid \Omega \mid \langle B; L\rangle \quad (1)$$

$$B ::= \epsilon \mid \alpha \mid \omega \mid v \quad (2)$$

$$L ::= \emptyset \mid d{:}C + L \quad (3)$$

where $d, v \in \mathbb{S}$.

There are three special contexts:

- $\bot$ is the *empty context* (also called *vanilla*);
- A is the *minimally defined context*, just more defined than the empty one;
- Ω is the *maximally defined context*, more defined than all other contexts.

The normal case is that there is a *base value* B, along with a *context list* (L for short), which is a set of *dimension-context* pairs. We write δL for the set of dimensions of L.

A sequence of dimensions is called a *compound dimension*. It can be used as a path into a context. Formally:

$$D = \cdot \mid d{:}D \quad (4)$$

If C is a context, $C(D)$ is the subtree of C whose root is reached by following the path D from the root of C:

$$C(\cdot) = C \quad (5)$$

$$\langle B; d{:}C' + L\rangle\ (d{:}D) = C'(D) \quad (6)$$

As with contexts, there are three special base values:

- ϵ is the *empty base value*;
- α is the *minimally defined base value*, just more defined than the empty base value;
- ω is the *maximally defined base value*, more defined than all others.

The normal case is that a base value is simply a scalar.

To the set $\mathbb{C}$, we add an *equivalence* relation $\equiv$, and a *refinement* relation $\sqsubseteq$. We begin with the equivalence relation:

$$\bot \equiv \langle \epsilon; \emptyset\rangle \quad (7)$$

$$A \equiv \langle \alpha; \emptyset\rangle \quad (8)$$

$$\Omega \equiv \left\langle \omega; \sum_{d \in \mathbb{S}} d{:}\Omega \right\rangle \quad (9)$$

$$\frac{L_0 \equiv_L L_1}{\langle B; L_0\rangle \sqsubseteq \langle B; L_1\rangle} \quad (10)$$

Thus, $\bot$ and A are notational conveniences, while Ω cannot be reduced. The normal case supposes an equivalence relation $\equiv_L$ over context lists:

$$\emptyset \equiv_L d{:}\bot \quad (11)$$

$$d{:}\langle B; L + L'\rangle \equiv_L d{:}\big(\langle B; L\rangle + \langle B; L'\rangle\big) \quad (12)$$

$$L \equiv_L \emptyset + L \quad (13)$$

$$L \equiv_L L + L \quad (14)$$

$$L + L' \equiv_L L' + L \quad (15)$$

$$L + (L' + L'') \equiv_L (L + L') + L'' \quad (16)$$

The $+$ operator is idempotent, commutative, and associative. Now we can define the partial order over entire contexts:

$$\bot \sqsubseteq C \quad (17)$$

$$C \sqsubseteq \Omega \quad (18)$$

$$\frac{C \neq \bot}{A \sqsubseteq C} \quad (19)$$

$$\frac{C_0 \equiv C_1}{C_0 \sqsubseteq C_1} \quad (20)$$

$$\frac{B_0 \sqsubseteq_B B_1 \quad L_0 \sqsubseteq_L L_1}{\langle B_0; L_0\rangle \sqsubseteq \langle B_1; L_1\rangle} \quad (21)$$

which supposes a partial order $\sqsubseteq_B$ over base values:

$$\epsilon \sqsubseteq_B B \quad (22)$$

$$B \sqsubseteq_B B \quad (23)$$

$$B \sqsubseteq_B \omega \quad (24)$$

$$\frac{B \neq \epsilon}{\alpha \sqsubseteq_B B} \quad (25)$$

$$\frac{v_0, v_1 \in \mathbb{S}_i \quad v_0 \sqsubseteq_i v_1}{v_0 \sqsubseteq_B v_1} \quad (26)$$

The last rule states that if v_0 and v_1 belong to the same set $\mathbb{S}_i$ and are comparable according to the partial order $\sqsubseteq_i$, then that order is subsumed for refinement purposes.

The partial order over contexts also supposes a partial order $\sqsubseteq_B$ over context lists:

$$\emptyset \sqsubseteq_L L \quad (27)$$

$$\frac{L_0 \equiv_L L_1}{L_0 \sqsubseteq_L L_1} \quad (28)$$

$$\frac{C_0 \sqsubseteq C_1}{d{:}C_0 \sqsubseteq_L d{:}C_1} \quad (29)$$

$$\frac{L_0 \sqsubseteq_L L_1 \quad L_0' \sqsubseteq_L L_1'}{L_0 + L_0' \sqsubseteq_L L_1 + L_1'} \quad (30)$$

Rule 30 ensures that the $+$ operator defines the least upper bound of two context lists.

Context and Version Domains. When doing intensional programming, we work with *sets* of contexts, called *context domains*, written $\mathcal{C}$. There is one operation on context domains, namely the *best-fit*. Given a context domain $\mathcal{C}$ of existing contexts and a requested context C_{req}, the best-fit context is defined by:

$$\mathit{best}(\mathcal{C}, C_{\text{req}}) = \max\{C \in \mathcal{C} \mid C \sqsubseteq C_{\text{req}}\} \quad (31)$$

If the maximum does not exist, there is no best-fit context.

Typically, we will be versioning *something*, an object of some type. This is done using *versions*, simply (C, object) pairs. Version domains $\mathcal{V}$ then become functions mapping contexts to objects. The *best-fit object* in a version domain is given by:

$$\mathit{best}_O(\mathcal{V}, C_{\text{req}}) = \mathcal{V}(\mathit{best}(\operatorname{dom}\mathcal{V}, C_{\text{req}})) \quad (32)$$

Context Operators. Context operators allow one to selectively *modify* contexts. Their syntax is similar to that of contexts.

$$C_{\text{op}} ::= C \mid [P_{\text{op}}; B_{\text{op}}; L_{\text{op}}] \quad (33)$$
$$P_{\text{op}} ::= -- \mid \epsilon \quad (34)$$
$$B_{\text{op}} ::= - \mid B \quad (35)$$
$$L_{\text{op}} ::= \emptyset_{L_{\text{op}}} \mid d{:}C_{\text{op}} + L_{\text{op}} \quad (36)$$

A context operator is applied to a context to transform it into another context. (It can also be used to transform a context operator into another; see below.) The $-$ operator removes the current base value, while the $--$ operator in P_{op} is used to clear all dimensions not explicitly listed at that level.

Now we give the semantics for $C\ C_{\text{op}}$, the application of context operator C_{op} to context C:

$$C_0\ C_1 = C_1 \quad (37)$$
$$\Omega\ C_{\text{op}} = \text{error} \quad (38)$$
$$\langle B; L\rangle\ [--; B_{\text{op}}; L_{\text{op}}] = \left\langle B; L\backslash(\delta L - \delta L_{\text{op}})\right\rangle\ [\epsilon; B_{\text{op}}; L_{\text{op}}] \quad (39)$$
$$\langle B; L\rangle\ [\epsilon; B_{\text{op}}; L_{\text{op}}] = \left\langle (B\ B_{\text{op}}); (L\ L_{\text{op}})\right\rangle \quad (40)$$

The general case consists of replacing the base value and replacing the context list. First, the base value:

$$B - = \epsilon \quad (41)$$
$$B_0\ B_1 = B_1 \quad (42)$$

Now, the context list:

$$L\ \emptyset_{L_{\text{op}}} = L \quad (43)$$
$$(d{:}C + L)\ (d{:}C_{\text{op}} + L_{\text{op}}) = d{:}(C\ C_{\text{op}}) + (L\ L_{\text{op}}) \quad (44)$$
$$L\ (d{:}C_{\text{op}} + L_{\text{op}}) = d{:}(\emptyset\ C_{\text{op}}) + (L\ L_{\text{op}}), \quad d \notin \delta L \quad (45)$$

Context operators can also be applied to context operators. There are two cases:

$$[P_{\text{op}}; B_{\text{op}_0}; L_{\text{op}_0}]\ [\epsilon; B_{\text{op}_1}; L_{\text{op}_1}] = \left[P_{\text{op}}; (B_{\text{op}_0}\ B_{\text{op}_1}); (L_{\text{op}_0}\ L_{\text{op}_1})\right] \quad (46)$$

$$[P_{\text{op}}; B_{\text{op}_0}; L_{\text{op}_0}]\ [--; B_{\text{op}_1}; L_{\text{op}_1}] = \left[--; (B_{\text{op}_0}\ B_{\text{op}_1}); \left((L_{\text{op}_0}\backslash(\delta L_{\text{op}_0} - \delta L_{\text{op}_1}))\ L_{\text{op}_1}\right)\right] \quad (47)$$

Now that we have given the formal syntax and semantics of contexts, version domains, and context operations, we can move on to typesetting.

The Running Context in Ω

As is standard, the abstract syntax is simpler than the concrete syntax, which offers richer possibilities to facilitate entry. Here is the concrete syntax for contexts:

C	::=	<>	Empty context
	|	~~	Minimum context
	|	^^	Maximum context
	|	<*val*>	Base value
	|	<*L*>	Subversions
	|	<*val*+*L*>	Base & subversions
val	::=	~	Minimum value
	|	^	Maximum value
	|	*string*	Normal value
L	::=	$dim{:}C\ [+\ dim{:}C]^*$	
dim	::=	*string*	

As for the context operation, here is the syntax:

C_{op}	::=	C	Replace the context
	|	[]	No change
	|	[val_{op}]	Change base
	|	[L_{op}]	Change subversions
	|	[val_{op}+L_{op}]	Change base & subs
val_{op}	::=	-	Clear base
	|	*val*	New value
	|	--	Clear subversions
	|	*val*+--	New base, clear subs
	|	---	Clear base & subs
L_{op}	::=	$dim{:}C_{\text{op}}\ [+\ dim{:}C_{\text{op}}]^*$	

In Ω, the current context is given by:

```
\contextshow{}
```

If D is a compound dimension, then the subversion at dimension D is given by:

```
\contextshow{D}
```

while the base value at dimension D is given by:

```
\contextbase{D}
```

This context is initialized at the beginning of an Ω run with the values of environment variables and command-line parameters. Once it is set, it can be changed as follows:

```
\contextset{C_op}
```

Adapting to the Context

During execution, there are three mechanisms for Ω to modify its behavior with respect to the current context: (1) *versioned execution flow*, (2) *versioned macros*, and (3) *versioned ΩTPs*.

Execution Flow. The new `\contextchoice` primitive is used to change the execution flow:

```
\contextchoice{{C_op_1}=>{exp_1},
               ...
               {C_op_n}=>{exp_n}
              }
```

Depending on the current context C, one of the expressions exp_i will be selected and expanded. The one chosen will correspond to the *best-fit* context among $\{C\ C_{\mathrm{op}1},\ \ldots,\ C\ C_{\mathrm{op}n}\}$ (see the discussion above of Context and Version Domains).

Macros. The Ω macro expansion process has been extended so that any control sequence can have multiple, *simultaneous* versions, at the same scoping level. Whenever *controlsequence* is expanded, the *most relevant*, i.e. the *best-fit*, definition, with respect to the current context, is expanded.

A version of a control sequence is defined as follows:

\vdef{C_{op}}*controlsequence args*{*definition*}

If the current context is C, then this definition defines the $C\ C_{\mathrm{op}}$ version of *controlsequence*. The scoping of definitions is the same as for TEX.

This approach is upwardly compatible with the TEX macro expansion process. The standard TEX definition:

\def*controlsequence args*{*definition*}

is simply equivalent to

\vdef{<>}*controlsequence args*{*definition*}

i.e., it defines the empty version of a control sequence.

As stated above, during expansion the best-fit definition of *controlsequence*, with respect to the current context, will be expanded whenever it is encountered. It is also possible to expand a particular version of a control sequence, by using:

\vexp{C_{op}}*controlsequence*

ΩTPs and ΩTP-lists. Beyond the ability to manipulate larger data structures than does TEX, Ω allows the user to apply a series of filters to the input, each reading from standard input and writing to standard output. Each of the filters is called an ΩTP (Ω Translation Process), and a series of filters is called an ΩTP-list.

There are two kinds of ΩTPs: internal and external. Internal ΩTPs are finite state machines written in an Ω-specific language, and that are compiled before being interpreted by the Ω engine. External ΩTPs are standalone programs, reading from standard input and writing to standard output, like Unix filters.

Internal and external ΩTPs handle context differently. For external ΩTPs, the context information can be passed on through an additional parameter to the system call invoking the external ΩTP:

program `-context=`*context*

Internal ΩTPs have been modified so that every instruction can be preceded by a context tag. Using the simplest syntax, this becomes:

<<*context*>> *pattern* => *expression*

When an internal ΩTP is being interpreted, an instruction is only examined if its context tag (defaulting to the empty context) is less than the current running context.

When ΩTPs and ΩTP-lists are being declared in Ω, the `\contextchoice` operator can be used to build versioned ΩTP-lists. These will be particularly useful for multilingual typesetting. See [3] for more details.

The Internals of the Typesetting Engine

The versioned macros and ΩTPs presented in the previous sections clearly facilitate the development of software that is more flexible, in the sense that if new parameters are added to a system, new code only needs to be written for those parts affected directly by the new parameters.

But it is still not clear how these mechanisms will help solve this problem of having multiple input and output formats for use with the same typesetting system, in particular, inside Ω.

We outline the solution here, since at the time of writing, we have not yet finalized the syntax.

Essentially, all of Ω's internal data structures will be made directly accessible to the user. These include at least:

- streams of characters;
- streams of glyphs;
- math lists;
- input to each ΩTP application;
- output from each ΩTP application;
- paragraphs;
- tables;
- pages;
- other kinds of boxes.

For each of these data structures will be specified a canonical serialization and deserialization. For each of the algorithms that can be applied to these data structures, a means for applying the algorithms from the user level will be defined as well.

As a result, it will be possible to completely manipulate what we might call the "canonical input" and the "canonical output" of the typesetter. Then inputting from different formats and outputting to different formats becomes much simpler. For input from a specific input format, an ΩTP-list must be defined to translate that input format into the "canonical input". For output to a specific output format, an ΩTP-list must be defined to translate from the "'canonical output" to the output format. These ΩTP-lists may well be parameterized by the current context to achieve specific results.

This approach is suitable for generic content that is found in all input and output files, but what about elements that are relevant for only specific kinds of input or output? For these elements, it is the versioned macros that will come into play. Some macros will have versions defined to deal with these elements only when the right conditions occur in the context.

Conclusions

Given the successful experimental work in generating MathML and XML directly from the Ω engine, we consider that the approach presented in this paper is both elegant and doable. In fact, we consider that this approach makes it possible to integrate into a single framework most of the existing extensions to TEX, and its original view that a document should be transformed into a DVI file that is in turn converted to (for example) PostScript.

However, the implications go well beyond just integrating existing work, even if that goal is both laudable and desirable. New possibilities will arise, since the user will no longer be forced to use the TEX document model, in which a document is just a series of pages. Ω then becomes usable for typesetting small pieces of text at a time, say to generate EPS files on-demand for the uses of other applications, such as online multilingual mapping tools or air traffic control systems.

For a piece of software to survive from one generation to the next, it must be able to adapt to the arising needs of coming times, and continually provide new possibilities. We believe that the approach presented here will ensure the long-term viability of Ω, hence of the TEX community.

References

[1] Donald Knuth. *Computers and Typesetting*. 5 volumes, Addison-Wesley, 1986.

[2] Omega Typesetting and Document Processing System, `http://omega.enstb.org`

[3] John Plaice, Yannis Haralambous and Chris Rowley. A multidimensional approach to typesetting. *TUGboat* 24(1), 2003, Proceedings of the TUG Annual Meeting, pp. 105–114.

[4] John Plaice and Joey Paquet. Introduction to intensional programming. In *Intensional Programming I*, World-Scientific, Singapore, 1996.

[5] John Plaice, Paul Swoboda and Ammar Alammar. Building intensional communities using shared contexts. In *Distributed Communities on the Web*, *LNCS* 1830:55–64, Springer-Verlag, 2000.

[6] John Plaice and William W. Wadge. A new approach to version control. *IEEE-TSE* 19(3):268–276, 1993.

[7] Paul Swoboda. *A Formalization and Implementation of Distributed Intensional Programming*. PhD thesis, University of New South Wales, Sydney, Australia, 2003.

[8] Extensible Markup Language (XML), `http://www.w3c.org/XML`

[9] Neomega Typesetting System. `http://neomega.web.cse.unsw.edu.au`.